The Source is the Proof

Vivek Haldar
Christian H. Stork
Michael Franz
University of California, Irvine

Acknowledgements

- Anonymous reviewers
- Some slides by Matthew Beers

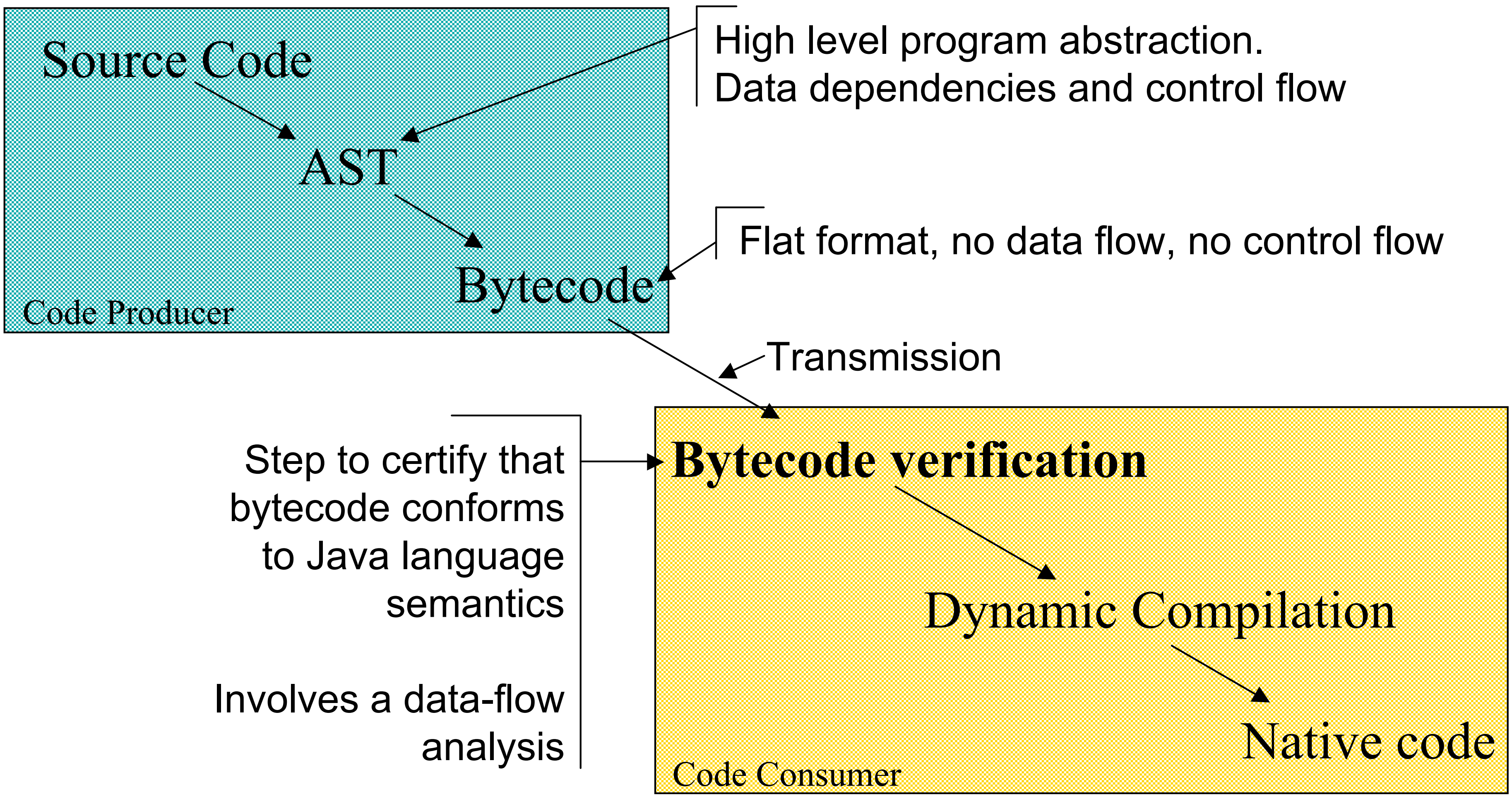
Outline

- Prevalent mobile code approaches - and its problems
  - Java bytecode
  - Proof-carrying code
- Our approach - compressed abstract syntax trees (CAST).

Mobile code wishlist

- Portable
- Secure
- Efficient
- Compact
- Language agnostic

Existing Process - Java bytecode



Java bytecode

- Large *semantic gap* between Java *language* and Java *bytecode*
- Much effort spent to ensure this gap is not exploited - *bytecode verification*

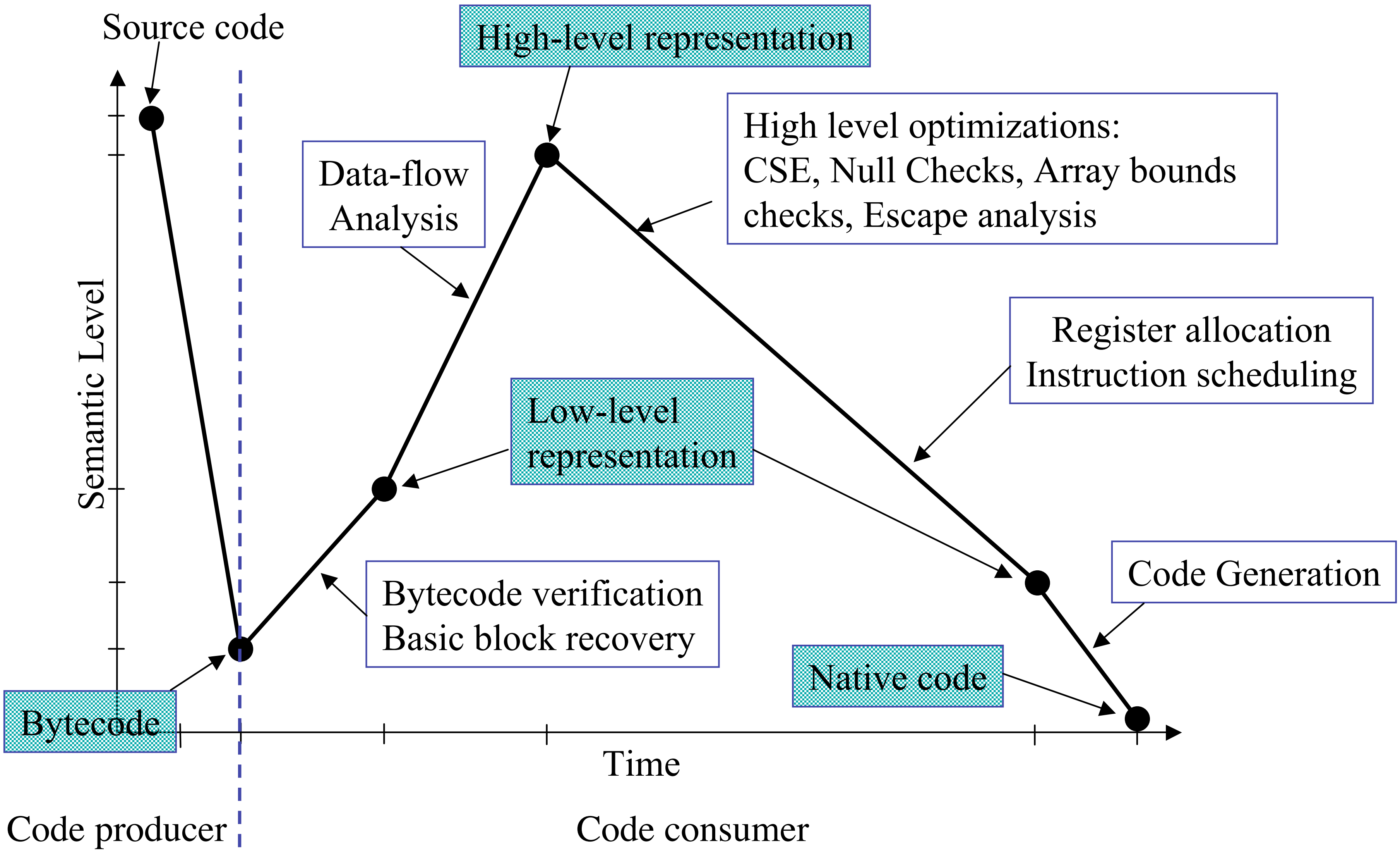
Verifying bytecode

- Why? Can do things in bytecode that are not possible in the Java language
  - Type-unsafe accesses
  - Illegal jumps
  - Using un-initialized objects
- Full data flow analysis
- Poorly specified, brittle
  - [Staerk] showed legal Java programs rejected by verifiers.

Certifying compilation

- Code carries along with it a proof/certificate of safety - PCC [Necula et al]
- Demonstrated with SafeC to Alpha assembly and Java bytecode to x86 - but not for a high level source language
- Not portable - certifies platform specific binaries
- Security policies fixed - must be known at proof-generation time.

Dynamic Compilation Process



Some common traits

- Very low level
- Very little high level semantic information
- ***Large semantic gap*** between properties being certified (type safety etc) and representation (bytecode)
- Hard to optimize

Why not use higher level representations?

Some questions

- Where do safety guarantees come from in a language?
- Why do we have to do so much work to recover them from our mobile code representation?

The Source…

- Consider a program in a high-level strongly typed language
- If this type checks, can make strong guarantees about its safety
- ***The source is essentially a proof of safety***
- Is lost/thrown away when using low-level mobile code representations

The Source is the Proof

A proof of safety exists at the source level - preserve this through the mobile code pipeline

Abstract Syntax Trees

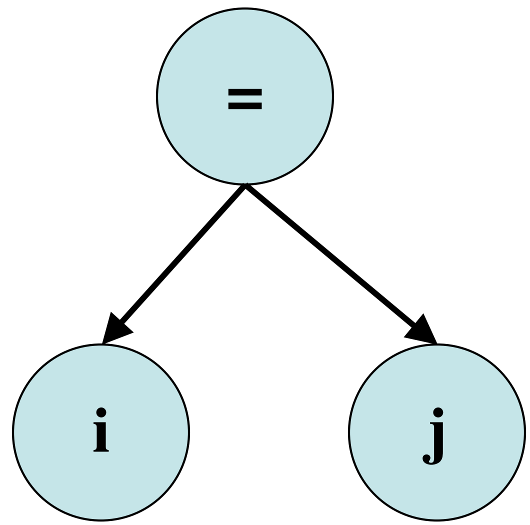
- Stripped down version of parse trees
- Semantically equivalent to source
- Governed by a grammar

Encoding ASTs

- How to encode ASTs in a way that safely transports source-level semantics?
- Enforce semantic constraints as an intrinsic part of the encoding itself
- *Wellformedness by construction*

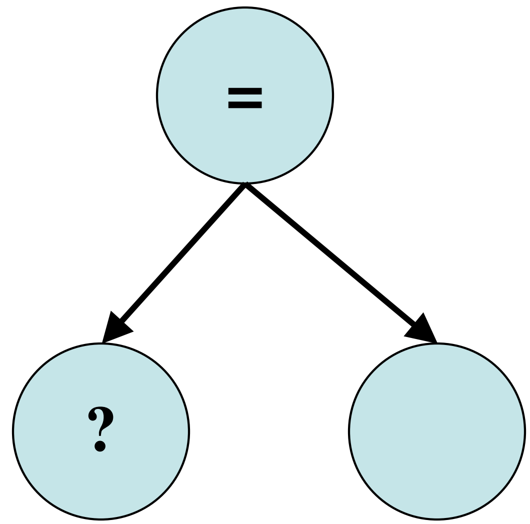
Example

int i, j, k;
char a, b, c;
j = 10;
i = j; // encoding this



Example

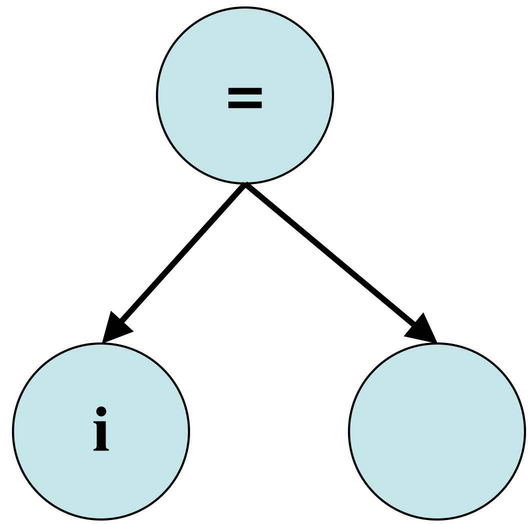
int ***i*, *j*, *k***;
char ***a*, *b*, *c***;
j = 10;
i = j; // encoding this



One of 6 choices

Example

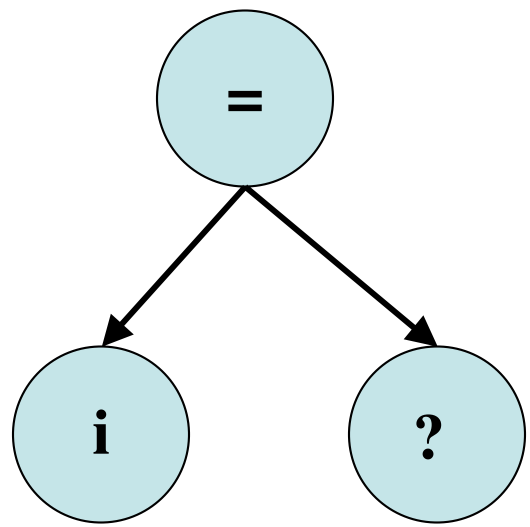
int ***i*, *j*, *k***;
char ***a*, *b*, *c***;
j = 10;
i = j; // encoding this



Encode index of choice taken

Example

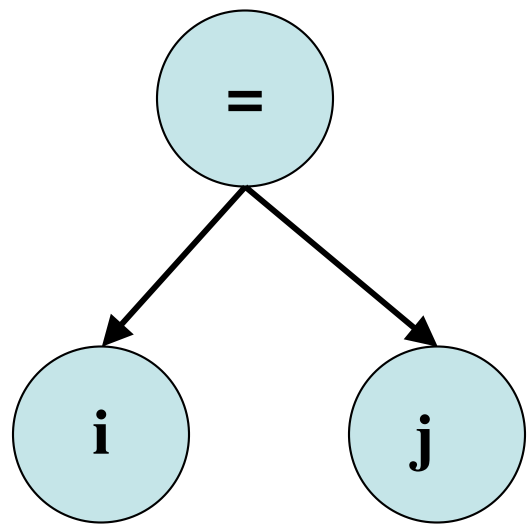
int ***i, j, k***;
char a, b, c;
j = 10;
i = j; // encoding this



One of 3 choices
Because of typing
rules

Example

int ***i, j, k***;
char a, b, c;
j = 10;
i = j; // encoding this



Encode index of
choice taken

Encoding ASTs

- Generate possible legal successors to current node
- Indicate index of node to encode
- Sometimes, only one possible successor - don’t need to encode anything
- Is *safe by construction* - i.e. impossible to represent illegal programs

- Upto four times as small as Java class files

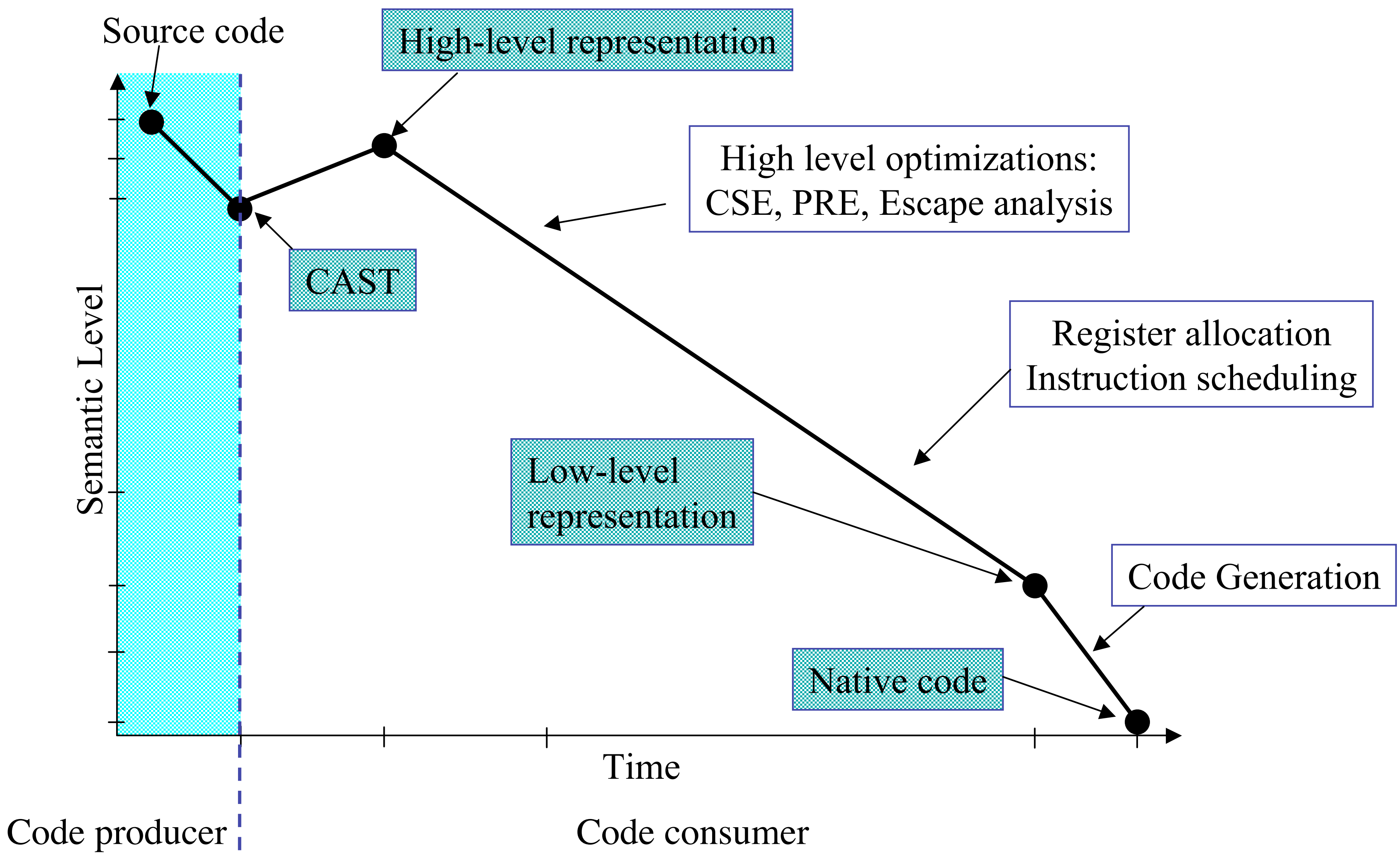
Adding annotations to ASTs

- ASTs very amenable to adding annotations - that are *verifiable*
- Hard to generate, easy to check
- E.g. implemented verifiable escape analysis annotatins by adding type modifiers to ASTs
  - 3% space overhead

An end-to-end mobile code system

- Prototype based on encoding Java source files
- Added verifiable escape analysis annotations
- GCC backend to native code generation

The New Process



Conclusion

- ASTs a viable alternative to low-level mobile code approaches
- Closer to source-level semantics
- Can transport annotations
- Can replace bytecode end-to-end
- Encoding is safe by construction, compact

Future Work

- Work on backend
  - Make use of escape analysis annotations
- More annotations
- Performance engineering

Do ASTs give away IP?

- Java disassemblers very good - e.g. Jode
- ASTs with scrambled variable names is very incomprehensible
- Impossibility of obfuscation [Goldreich et al]

PCC proofs as trees

- Recent work to address size of PCC proofs [Necula 2000]
- Views proofs as trees
- Uses predictive compression
- Narrows possibilities at “choice points” - indicates which choice is taken
- Very similar to how we encode ASTs